Batch Tasks User's Guide

I. P. Sharp Associates

SHARP APL/370
Release 19

APL Systems Development Group
I. P. Sharp Associates Limited
2 First Canadian Place, Suite 1900
Toronto, Ontario, Canada.
M5X 1E3

Telephone 416 364-5361
Telex 06-22259 I P SHARP TOR

PREFACE

This book replaces "Batch Tasks in SHARP APL" written in 1978 by PCB, printed again in 1986.

Related Publications:

These I.P. Sharp publications are referred to in this book:

SHARP APL/370 Usage Inquiry System User's Guide
(code: 0132 8703 E5)

SHARP APL Reference Manual (code: 79RM05)

Printed in Canada
March 1987
Publication Code: 0084 8703 E3

CONTENTS

1 INTRODUCTION, 1
- Overview, 1
- Advantages and Disadvantages of Batch Tasks, 1
- The Four Main Kinds of Tasks, 2
  - T-task, 2
  - S-task, 2
  - N-task, 2
  - B-task, 3
- Specifying the Batch Task Workspace, 3
  - Autostart, 3
  - Split Workspace, 4
- Passing Initial Data to an N-task or B-task, 4
- Communication between Tasks, 5
- Files Tied by Batch Tasks, 5

2 THE OWNERSHIP OF A BATCH TASK, 7
- Who Can Start a New Task?, 7
- Who Owns a Batch Task?, 7

3 STARTING AN N-TASK OR B-TASK, 9
- The Control Statement, 9
- Some Sample Arguments for □*RUN*, 11
- The Result of □*RUN*, 12

4 SUBMITTING A BATCH TASK REQUEST, 15
- Introduction, 15
- The Information Needed for a Batch Task Request, 15
- Batch Task Processing Specs, 17

5 PREPARING A BATCH TASK, 21
- Introduction, 21
- Output Generated by a Batch Task, 21
- Printing Data Previously Sent to a □*OUT* File, 22

6 MONITORING THE PROGRESS OF A BATCH TASK REQUEST, 23
- Introduction, 23
- Checking the Status of Your Batch Task Request, 23
  - Fields of *BTINQ*, 24
  - Results of *BTINQ*, 25
- Displaying a Summary Report of Inquiries, 26
- Withdrawing a Batch Task Request, 28
- Checking the Status of an Active Task, 28
  - Columns in □*RUNS*, 29
  - Sample Use of □*RUNS*, 29
- Identifying the Task-id of the Active Task, 30
- Interrupting a B-task or an N-task, 31
- Account Information □*AI*, 31

7 STARTING -- AND RESTARTING -- A BATCH TASK, 33
Factors in the Design of the Latent Expression, 33
Ingredients of the Latent Expression for a Batch Task, 34
A Sample Latent Expression for a Batch Workspace, 35
Determining Why a Batch Task Terminated, 37
File Synchronization Following a System Crash, 37

8 INDEX, 39

CHAPTER 1: INTRODUCTION

OVERVIEW

The term "Batch Tasks" refers to two different ways in which you can get the APL system to run an APL program in a workspace that runs automatically, without your direct control from a terminal. They are called "batch" tasks because you have to prepare their work in advance, rather than controlling it, step by step, as you go along. The two types of batch task are called 'N-tasks' and 'B-tasks'. When such a task is started, it runs unattended. Such unattended tasks are charged at a lower rate than applies to tasks that you control directly. An N-task starts at the time specified by the requestor. In contrast, a B-task is put into a queue of work which is processed at a later time, as dictated by system load. B-tasks are charged at a lower rate than N-tasks.

Preparing a program to run unattended may require a few changes from the way you might otherwise write it, mainly because there's no terminal at which to print answers or from which additional information can be supplied. But apart from that, batch APL is just like APL at the terminal. In particular, the language is the same, and, just as at a terminal, all work is done in an active workspace.

SHARP APL provides means to start tasks and to make inquiries about their status. This manual describes those facilities, and includes some advice about setting up programs to run in batch mode. These APL facilities are also described in the "SHARP APL Reference Manual."

ADVANTAGES AND DISADVANTAGES OF BATCH TASKS

The advantage of scheduling unattended tasks is that, in a situation where it isn't important to have immediate input/output at the terminal, you can get your work done without tying up a terminal. At the same time, you also save the connect charges, the communication charges (if any), and the character charges for terminal input and output. If you are willing to accept delayed execution -- for example, within a few hours, or overnight -- your computing is billed at a lower rate. For a job involving heavy computation, this can be an important saving.

The main disadvantage of using an unattended task is that you can't see what's happening. When the system finds an error in your program, it can't immediately send you an error message and await your correction. So batch work should be reserved for programs that you're confident can run unattended.

By running without the use of a terminal, you don't give up the ability to produce written output. It's a simple matter to have the output directed to a file, from which it can be displayed at a terminal later, or sent to a highspeed printer. If highspeed printer output is a convenient alternative for you, there are further savings possible, since in most instances it costs less to produce the same material at a printer than it would at a terminal.

THE FOUR MAIN KINDS OF TASKS

In the SHARP APL system, all computation takes place in an active workspace. The work session of an active workspace is called a 'task'. When you sign on at a terminal, the first part of the system acknowledgement you receive is the task number, or task-id, of your session.

Tasks are of several kinds, only two of which are described in this manual. The four principal types are:

T-task

The task consists of all the work done in your active workspace from the time you sign on at a terminal until you sign off. This is the type of task with which you're most likely to be familiar. A T-task is normally in immediate execution mode. That is, it is waiting for an instruction from the keyboard. When it receives it, it carries it out at once, and then waits to receive another instruction.

S-task

An S-task is, in most respects, identical to a T-task. However, instead of receiving input from a terminal and directing output to a terminal, it communicates instead with another task by way of a shared variable.

N-task

An N-task is a new active workspace which starts to execute immediately after it is created. When it has finished executing the program with which it started, it stops work; its active workspace is saved and it is signed off. An N-task has no provision for input of data or further instructions from a terminal, or for output to a terminal. Because there's no return to immediate execution, and no means to receive new instructions to be executed, whatever the N-task does must be able to run unattended. If the N-task requires input to work on, or must report its results, it must do those by using files, or by using shared variables to communicate with another task which provides the input or receives the output. When work is ended, the N-task's active workspace is automatically saved under a name you specify at the time you start the N-task.

If an error is encountered during work on an N-task, the system can't type out an error message and wait for a correction, the way

it could during a T-task or an S-task. If something goes wrong, work on the N-task is halted, and its active workspace is saved. Later, by looking at the state of the workspace, you'll be able to see what was being executed when work was abandoned.

An N-task may be started in one of two ways. You can use the system function ⎕*RUN* to directly start the N-task, which is referred to as a 'direct N-task'. Alternatively, you might use a system facility called the Batch Task Scheduler to start the N-task, which is called a 'scheduled N-task'. You would choose the latter if you wished to take advantage of facilities provided by the Batch Task Scheduler, such as task restart. Regardless of the method used, the instant at which the N-task starts is determined by you. See the section "Starting an N-task or B-task" for more details.

B-task

A B-task is like an N-task except that:

- To start a B-task, you must always submit a request to the Batch Task Scheduler.
- The time at which a B-task starts is determined by the Batch Task Scheduler, based on its own considerations of system resources and load. The time that the scheduler starts your task may be a considerable period after the request was submitted.

Apart from the way they handle input, output, and errors, all four types of tasks use APL in the same way. You can always take a workspace that was intended to run in batch and run it as a T-task. Conversely, except for terminal input/output, a program developed as a T-task can be run as an N-task or B-task. In fact, that's probably the way most batch tasks are initially developed.

SPECIFYING THE BATCH TASK WORKSPACE

There are two ways to specify the workspace a batch task is to use: the **autostart** workspace and the **split** workspace. Autostart can be used both for N-tasks and for B-tasks. Split workspace can be used only for direct N-tasks.

Autostart

To specify an autostart workspace, you provide the name of a saved workspace. The system loads that workspace, and executes its latent expression, ⎕*LX*. Executing that expression should set in motion all that you want done. It's up to you to see that the workspace you've named has been stored, and contains appropriate programs and an appropriate ⎕*LX*. When the work comes to a halt, the workspace is saved under a name you provided as part of the original request.

Warning: There are some cautions to be observed in setting up the latent expression for an autostart workspace. Don't try it until

after you've read the sections entitled "Starting -- and Restarting -- a Batch Task."

Split Workspace

You indicate that an N-task is to start from your current **active** workspace. The N-task thus created continues executing whatever function was being executed in the active workspace at the moment when the N-task was created.

The system creates a duplicate of your current active workspace, exactly as it is, and keeps right on executing whatever function was being executed. (A split workspace does not involve loading a stored workspace and does not execute $\square LX$.)

Warning: The new N-task goes on executing the very same program that was executing in the active workspace that invoked it. You have two active workspaces doing the same thing. For this to make sense, the program must contain a test that identifies which task it belongs to, so that, once the new task is split off, it doesn't just duplicate the work of the parent task, but does something else more useful. Don't try running a split workspace until after you've read the sections entitled "The Control Statement," and "Factors in the Design of the Latent Expression."

PASSING INITIAL DATA TO AN N-TASK OR B-TASK

One of the more convenient ways to pass initial data to an N-task or B-task is to use the session parameter $\square SP$. Before you start the batch task, put into $\square SP$ the data the task will need. Arrange the batch task so that it uses $\square SP$ as the source of its data.

The session parameter $\square SP$ from the current workspace is carried forward into the active workspace of a direct N-task; this is true both in autostart and split-workspace cases. $\square SP$ is not automatically carried into a batch task, but you may include, in your batch task request, provision to have the Batch Task Scheduler copy your present $\square SP$ and later recreate it in the workspace of your batch task at the time the scheduler runs it. You should also be aware that a direct N-task receives $\square SP$ from its parent workspace whether it is useful or not; $\square SP$ might conceivably take up space that the N-task will need for something else.

COMMUNICATION BETWEEN TASKS

All the work that a batch task does must flow from the latent expression, if it's an autostart workspace, or from what it was doing at the moment it was spawned, if it's a split-workspace N-task. As soon as a batch task returns to immediate execution, it immediately halts; there's no way to give it a new instruction to execute. But an N-task **can** continue to exchange data with its parent task, or with any other tasks, provided the other tasks are running at the same time. The tasks communicate by sharing one or more variables between them. For example, a function in the parent workspace may start an N-task containing the programs needed for a particular portion of the work, and then use shared variables to transmit to the N-task the particular data to be worked on, and receive the results by the same route. This doesn't have to be a single exchange; a protracted dialogue is possible.

It is also possible to design an N-task which serves not one user, but many, acting itself as a time-sharing service in miniature, responding to requests sent to it from any number of other tasks, and sending back to each the result it has calculated for it. This manual does not cover techniques for using shared variables. You should consult the "Sharp APL Reference Manual."

A B-task could also share variables with another task, however the task with which it shares must be running at the same time, and you cannot anticipate when the B-task is going to run.

FILES TIED BY BATCH TASKS

A batch workspace, whether B-task, autoload N-task, or split-workspace N-task, starts with **no** tied files. Note that a workspace spawned from your current active workspace may contain references to files which were tied in the parent workspace, but are no longer tied in the spawned workspace. If you need to keep on using those files, you must include provision to tie them. If both tasks will use the same files at once, you must share-tie rather than exclusive-tie them. You may also need to hold files if both tasks make updates.

CHAPTER 2: THE OWNERSHIP OF A BATCH TASK

WHO CAN START A NEW TASK?

You can start a T-task only by going to a terminal and (manually) signing on. You can only have one T-task signed on to a given account number at a time.

An N-task can be started, or a Batch task request can be submitted, by any active task. Since you always have to start out at a terminal, your first batch task (N-task or B-task) must be started from your T-task. But from then on, any N-task or B-task, once running, can start another N-task or B-task. (Any active task can start an S-task, but they are outside the scope of this manual.)

The system imposes a limit on the grand total of N-tasks that it can handle at one time (from all users), and also on the maximum number of tasks, other than B-tasks, that any one user can have going at once. B-tasks are not included in this count because you cannot predict when B-tasks are started. On the I.P. Sharp Toronto system, the maximum number of tasks for one user is currently 9, and the maximum number of batch task requests one user can have queued at one time is currently 21.

WHO OWNS A BATCH TASK?

Every task running on the APL system has its own unique task-id. The task-id of your T-task is one of the items displayed when you sign on. The fact that each task has a unique number is one way that you can distinguish an N-task that has been split off from another active workspace: the new N-task has a task-id different from the parent task by which it was created.

Every active workspace has an owner; that is, it is associated with a particular user account that determines what the active workspace is entitled to do (for example, what files it can access). The user account gets the bill for the work the task does.

Ordinarily, an N-task or a B-task is owned by the account that created it. However, it is possible for a task of yours to start a task that will belong to someone else, or conversely, for someone else to start a task that will belong to you. Starting an N-task or a B-task is thus like signing on: if you have been provided the account number and the current password of another account, then you can start an N-task or a B-task belonging to that other account. The N-task or B-task will then have whatever powers (or limitations) go with that account number. Any reference to the name of a workspace or a file, unless explicitly qualified by account

number, is presumed to refer to the library of the account that owns the task.

Each task is associated with:

- its own task-id
- the account number under which it is running, or its owner
- the account number of the task that started it

The last two items are different when a task belonging to one account started a task on behalf of another account.

THE CONTROL STATEMENT

To submit a batch task, you have to provide five facts. These are represented in a character vector. There are two ways to use that character vector:

1. If you're starting an N-task directly, the character vector is the argument of the system function ⎕*RUN*.

2. If you're submitting a B-task or scheduled N-task request, that character vector is passed to the Batch Task Scheduler by functions that are to be found in the public workspace 1 *BTASKREQ*.

In either case, the same five facts are required. The following is a sample use of ⎕*RUN* to start an N-task:

```
⎕RUN '1234567:PASSWD LOADWS:LOCK1 SAVEWS:LOCK2 10 1000'
```

The character vector is divided into five fields, separated by blanks, as described below:

Account number
: The account number, a colon, and the password of the account that is to own the task. In the preceding example, the sign-on was:

```
1234567:PASSWD
```

You need to specify a sign-on only if you're starting a task to run on someone else's account. If the task is to run on your own account, you can just put a colon, although you can include your own sign-on explicitly if you want to.

Load-workspace name
: This is the name of the workspace that is to be loaded to start the task. If the task you're starting is a B-task or scheduled N-task, this information must be provided, since such a task always starts from a saved workspace. But if you are starting an N-task directly by splitting the active workspace, put only a colon here.

If you simply write the name of a workspace without an account number, the workspace is presumed to belong to the account identified in the first field. If the starting workspace is to be loaded from some other library, you must include the account number ahead of the workspace name, just as you would if you

were loading that workspace. If the workspace is locked, you must include a colon and its lock.

Save-workspace name
This is the name under which the task's active workspace is to be saved when work is completed. This is sometimes called the **continue-workspace**, from analogy with the *CONTINUE* workspace saved when a T-task is interrupted.

This field can never be elided; you must name a workspace here. Work on an N-task or a B-task always ends by saving the active workspace, unless some system failure has made that impossible, or the active workspace has terminated in a clear workspace (for instance after executing the function *CLEAROUT*). The workspace is saved either at the normal end of work, or at the point when work was interrupted (because the processing limits were exceeded, because you interrupted the task using the system function □*BOUNCE*, or because processing was abandoned after an error was encountered).

Notice that the save-workspace can only be saved into the library of the account that owns the task. As with the load-workspace, that account number is understood and so does not need to be written explicitly. However, if you choose to mention an account number for the save-workspace, it must be the number of the owner of the task.

Warning: If you have already saved your entire quota of saved workspaces, you won't be able to save another, and the system will reject your attempt to start an N-task or B-task.

CPU limit
This is the maximum number of CPU units to be permitted for the task. When processing reaches this limit, execution is halted and the workspace is saved, whether or not work has been completed.

If you set this limit too low, your job can't be completed. If you set it too high, you run the risk of, if your program contains an error causing it to run indefinitely, running up an excessive bill before the system calls a halt.

Connect limit
This is the maximum connect seconds. When processing reaches this limit of elapsed time, execution is halted and the workspace is saved, whether or not work has been completed.

Here again, setting the limit too low will prevent your job from completing. Setting an unrealistically high limit when you're submitting a B-task gives rise to another problem: the Batch Task Scheduler may needlessly delay running your job, because the job looks more demanding than it really is. The scheduler will only start your B-task when the time you have requested appears to be available, or you have indicated that it's all right for the system to halt your task while it's running and restart it later. See the discussion of *NORMRST* in the section entitled "The Information Needed for a Batch Task Request."

Warning: If you use a limit of 0, the system interprets it as "no limit". A limit of 0 is permissible but dangerous. If your program contains an endless loop, it could run a very long time before someone stops it. A B-task whose time limit is 0 **must** include the specification that it can be restarted after shut-down (*NORMRST*).

Some Sample Arguments for □*RUN*

```
□RUN ': MYWS END 10 600'
```

In the character vector shown above, the initial colon means that the sign-on is elided. Hence the account under which this statement is executed is also the owner of the N-task. This example is probably the most common situation.

The request is to create an N-task by autostarting the workspace called *MYWS*, and, when work is ended, to save the active workspace under the name *END*. A CPU limit of 10 units and a connect limit of 600 seconds is set.

```
□RUN ': 75461 ACCT:COST FINAL:SECRET 200 8000'
```

The character vector shown above is a request to start an N-task that will run under the same account as the task that executed this statement. It requests the creation of an N-task by autostarting a workspace from library 75461 called *ACCT*, which is locked with the password *COST*. At the end, the active workspace is to be saved under the name *FINAL* and locked with the password *SECRET*. Limits of 200 CPU units and 8000 seconds are set.

```
□RUN '1234567:XYZ 75461 ACCT:COST END:ULT 200 8000'
```

The character vector shown above, which includes an explicit sign-on in its first field, requests that an N-task be started on behalf of user 1234567, whose sign-on password is *XYZ*.

The request is to create an N-task by autostarting the workspace 75461 *ACCT*, which is locked with the lock *COST*. If the number 75461 were not shown in front of the workspace name *ACCT*, *ACCT* would be presumed to belong to library 1234567. At the end, the active workspace is to be saved under the name *END* and locked with the lock *ULT*. Note that since this work is to be under account 1234567, the workspace *END* will likewise belong to library 1234567, and not to the library of the account that submitted the request.

```
□RUN ': : END 80 1800'
```

In the character vector shown above, the second field, load-workspace, is elided and appears only as a colon. This vector is a request to start an N-task by splitting the active workspace. In a split workspace request, the load-workspace is always elided, and appears only as a colon. The colon in the

first field indicates that the sign-on is elided also, so the spawned N-task will be owned by the same account as the originating task.

When work on the spawned N-task is ended, its active workspace is to be saved under the name *END* (without a lock). That workspace will be saved into the library of the account that owns this task. In this case, since the sign-on was elided, that is the library of the account that submitted the request. If some other account had been identified in the sign-on, the save-workspace called *END* would be saved in that account's library.

The Result of □*RUN*

When you execute □*RUN* with some suitable argument you get a two-element numeric vector as the result of □*RUN*.

The first element indicates whether or not your attempt to start an N-task was successful. The value 0 means it was. Any other value in the first element is an indication of the kind of error that was encountered.

The second element is sometimes used to convey additional information. If your request was successful, the second element is the number of the new N-task you just started. If there was an invalid argument to □*RUN* (indicated by a 1 in the first element of the result) then the second element is the position in the argument at which the error was detected. In all other cases the second element of the result is 0.

Notice that it is the responsibility of the program that requests an N-task to verify whether the request has been successful. A failure is reported only by the first element of the result of □*RUN*. This situation is unlike many errors in APL; in many, an error causes work to be interrupted and an error message displayed. Failure of a □*RUN* shows up only in the value of its result.

If the request was to split the active workspace, it is also the responsibility of the program that makes the request to test the result of □*RUN*, or make some other test, in order to distinguish the task that made the request from the new N-task that has been spawned.

The various values that can result from □*RUN* are:

0	0	The workspace in which this result is obtained is a newly-spawned split-workspace N-task. (The workspace that requested the N-task receives a result in which the second element is the task-id of the task just started.)
0	**n**	The request was successful, and an N-task with task-id **n** has been started.

1	n	The argument to □*RUN* was found to be defective at a position **n** characters in from the left end.
2	0	The system can't handle any more N-tasks at the moment. (There may be nothing wrong with your request as stated; it's just that system resources are exhausted.)
3	0	The account number attempting to start the N-task has already started its quota of tasks. This limit is site defineable, and is currently 9 on the IPSA timesharing system.
4	0	The account which was to own the requested N-task does not have a workspace quota sufficient to permit the save-workspace to be saved. (The system checks this before starting the task.)
10	0	The sign-on account number or password is invalid.
11	0	The sign-on account number is locked out.
12	0	The load-workspace cannot be found, or has a lock different from that included in the request.
13	0	The load-workspace did not contain a latent expression. This could not apply to a split workspace.
14	0	System resources are exhausted. No more tasks can be handled at the moment.
15	0	The user who submitted the request is not allowed to use □*RUN*.
16	0	A hardware read error occurred during an attempt to load the load-workspace.
17	0	The account number attempting to start the N-task is not allowed to do so on the owner account number.

These values appear as the two-element result of □*RUN*, or as columns 3 and 4 of the result of an inquiry about the disposition of a batch task request made with the function *BTINQ* in workspace 1 *BTASKREQ* (see "Monitoring The Progress of a Batch Task").

CHAPTER 4: SUBMITTING A BATCH TASK REQUEST

INTRODUCTION

The public workspace 1 *BTASKREQ* contains functions for submitting a batch task request. The functions exist in two forms: interactive and automatic. Either way, the same information is required, but the interactive version prompts you for the various items it requires.

Each successfully submitted batch task request is assigned a request number. This isn't a task-id; your task hasn't yet been run. It's the number of your request in a file of requests maintained by the scheduler. Once you have the request number, you can make inquiries regarding what the Batch Task Scheduler has done with your request.

The syntax of the two functions for submitting a batch task request is:

```
N←BTASKREQ
```

whereupon the function prompts you for the information needed; or

```
N←BTSUBMIT X
```

where *X* is a character vector containing your entire request. In either case, the result returned is the number assigned to your request by the Batch Task Scheduler.

To use *BTASKREQ* or *BTSUBMIT*, you can either load the workspace 1 *BTASKREQ*, or copy the function you want from that workspace into your own.

THE INFORMATION NEEDED FOR A BATCH TASK REQUEST

The information needed to submit a request is of two kinds. These are called the *⎕RUN* **parameters** and the **processing specifications**. The *⎕RUN* parameters are identical to those for the argument of *⎕RUN* used to request an N-task directly. The processing specs give the Batch Task Scheduler additional information about when your task can be scheduled and whether or not it can be restarted (using the original or continue workspace) if interrupted by a normal system shutdown or a system crash.

If you use *BTSUBMIT*, its argument is a character vector identical to the one you would use for *⎕RUN*, preceded, if appropriate, by

the processing specs. If there are no processing specs, then the argument is exactly the same as for ⎕*RUN*. But if there are some processing specs, they come first in the vector, and are separated from the remainder by the jot symbol (∘). The various possible processing specs are described in the next section.

Here's a sample *BTSUBMIT* with no processing specs:

```
      N←BTSUBMIT ': LOADWS:LOCK1 SAVEWS:LOCK2 10 1000'
```

Below appears the same request, but with the addition of processing specs which state that the task can be restarted, using the original workspace, if interrupted by a normal system shutdown, and that it must not be run before 17:30 hours:

```
      N←BTSUBMIT 'NORMRST(LOAD), NBTIME(17:30)∘: LOADWS
                 :LOCK1 SAVEWS:LOCK2 10 1000'
```

If you prefer to be prompted for this information, you can use the function *BTASKREQ*. It works either in a terse form, which asks two questions, first for the processing specs and then for the ⎕*RUN* parameters, or in a loquacious form, in which it asks separately for each of the five ⎕*RUN* parameters.

Either way, you use the same function *BTASKREQ*. If your answer to its second prompt (requesting the ⎕*RUN* parameters) is simply a carriage return, it proceeds to prompt you for each of the five ⎕*RUN* parameters in turn.

Below appears an example showing the same request illustrated above, but with the terse form of *BTASKREQ*:

```
      N←BTASKREQ
PROCESSING SPECS - NORMRST(LOAD) NBTIME(17:30)
⎕RUN PARAMETERS  - : LOADWS:LOCK1 SAVEWS:LOCK2 10 1000
```

Below appears the same request, but this time entered with an empty answer to the second prompt, so that the function then prompts for each of the ⎕*RUN* parameters individually:

```
      N←BTASKREQ
PROCESSING SPECS - NORMRST(LOAD) NBTIME(17:30)
⎕RUN PARAMETERS  - carriage return
SIGNON           - :
LOAD WSID        - LOADWS:LOCK1
SAVE WSID        - SAVEWS:LOCK2
CPU LIMIT        - 10
CONNECT LIMIT    - 1000
```

The function *BTASKREQ* returns a message showing the number that has been assigned to your batch task request, and the time and date.

Batch Task Processing Specs

There are several processing specifications available to all users. (If you have an application that could benefit from a special processing spec different from these, it may be possible to arrange it by contacting the data centre, or your system administrator.) Each of the standard processing specs consists of an abbreviation either standing alone, or followed by an argument enclosed in parentheses. When an argument in parentheses is included, it must immediately follow the abbreviation without intervening blanks. When you use more than one specification in the same request, the various specs are separated by blanks or commas. The current options are as follows:

NORMRST([*LOAD*][*CONTINUE*])
Restartable, using the original or continue workspace, after interruption by normal shutdown.

If the task is interrupted by a normal system shutdown, it can be restarted using either the original workspace, or a continue workspace, as specified. Restarting with the original workspace implies that the latent expression is such that, when the workspace is autostarted a second time, its behaviour will be appropriate. If no argument is included with *NORMRST*, *CONTINUE* is assumed.

During a normal shutdown, a job in progress is halted by a weak interrupt (like pressing the BREAK or ATTN key once at a terminal), so that you are reasonably assured that the halt took place at the end of a statement, and not in the middle.

NORMRST **must** be included in a request if a connect limit of 0 is specified in the □*RUN* parameter portion of the request.

ABNORMRST([*LOAD*][*CONTINUE*])
Restartable, using the original or continue workspace, after interruption by system crash.

The task can be restarted using either the original workspace or a continue workspace, as specified, after an abnormal end (i.e., after a system crash). Restarting with the original workspace implies that the workspace has a latent expression constructed so that when the workspace is loaded a second time, its behaviour is appropriate. If no argument is included with *ABNORMRST*, *CONTINUE* is assumed.

A system crash differs from normal shutdown in two regards. File system operations may be out of synchronization with the continue workspace. There is a greater possibility that the current line will be interrupted part way through execution. Thus, selecting this option implies either that you are confident that the code can be restarted wherever it left off, or that you have provided some means for the program to transfer to a point at which work can safely be resumed. Restartability is discussed in the section entitled "Starting -- and Restarting -- a Batch Task."

It is not always certain that the system will be able to save a continue workspace for your task after a crash. The crash may have been caused by the very problem that lost your continue workspace. As a result the chances of restart after a crash are improved if you specify the (*LOAD*) parameter. Of course, if the original load workspace no longer exists, perhaps as a result of)*DROP*, restart will fail.

NBTIME(*HH:MM*)
Not before indicated time of day.

The batch task should not be scheduled before the indicated time of day. Time is SHARP APL system time, which may be different from your local time. Whether you specify *NBDATE* or not, the time meant is presumed to be a time on whatever day the scheduler tries to initiate the batch task. For example, if the scheduler is unable to create the batch task on a particular day, it will try again but not before the designated time on the next day. A period or a blank can be used instead of a colon to separate hours from minutes. Leading zeros are not necessary in the hours or minutes fields.

NBDATE(*MM/DD/YY*)
Not before indicated date.

The batch task should not be scheduled before the date shown. Time and date are according to SHARP APL system time, which may differ from your local time. A blank can be used instead of the slashes. Leading zeros in the month and day fields are not necessary.

If you use both *NBDATE* and *NBTIME*, you should be aware that there is interaction between them. The request will only be scheduled after the *NBTIME* value, even if the *NBDATE* value has past many days ago.

NBTS(*YYYY/MM/DD HH:MM*)
Not before indicated time and date.

The batch task should not be scheduled before the time and date shown. The timestamp is in the same format as the □*TS* system function. For example, *NBTS*(1986/07/13 14:00) specifies that the task is not to be started before 14:00 on July 13th, 1986. The century and the leading zeroes in the date and time fields are optional. Blanks may be used in place of slashes and colons to delimit fields.

NBBTASK(*NNN*)
Not before termination of the batch task with the indicated request number.

The batch task should not be scheduled until after completion of the specified batch task. If the indicated batch task can't be started, this will be indicated by the value 60 0 in the result of the function *BTINQ* (see "Results of *BTINQ*").

Notice that the batch task need not have completed its work successfully, but only to have ended its work. If work ended

because an error was encountered, that would still meet the requirements of this specification. If you want to be certain that the specified task has completed its work successfully before starting this request, you should have the other task submit this request as its last step, following all its other work.

If you want your request held until after several other requested tasks have all been run, include *NBBTASK* several times in the list of specs, once for each task. For example:

```
NBBTASK(459),NBBTASK(463)
```

NBSORT(NN)
Not before successful termination of the sort with the indicated request number.

The batch task should not be scheduled until after the specified sort request has successfully ended. The requested batch task is not run if the sort was not run, or if an error was encountered during its execution. If the indicated sort can't be completed, this will be indicated by the value 61 0 in the result of the function *BTINQ* (see "Results of *BTINQ*)

As with *NBBTASK*, this spec can be used more than once in the same request.

NBOR(TEXT)
Not before operator releases (according to indicated message).

The text is a message in which you request the manual intervention of the system operator to assure that some condition has been met before your batch task is scheduled. For example, if your batch task is to process data from an imported data file which has not yet been put up as an APL file, you might include a message such as this:

```
NBOR(PLS VERIFY TAPE XYZ CONVERTED ∧ MOUNTED AS
PER MBX MSG 503011)
```

COPYSP
Copy requestor's □*SP* value forward into the batch task workspace.

Specifies that the value of □*SP* in the requestor's workspace is to be propagated into the batch task workspace on initial startup of the batch task. If *COPYSP* is not included in the processing specifications, the default value of □*SP* (an empty character vector) is propagated into the batch task workspace. This default value is always propagated into the workspace when a batch task is restarted after a system shutdown or crash.

NTASK[(RETRY)]
Run as an N-task.

The job is run as an N-task and is subject to all normal system and user N-task limits and billing criteria, as with N-tasks submitted directly with □*RUN*. The N-task is started by the Batch

Task Scheduler as soon as all processing specifications are satisfied.

If the Batch Task Scheduler cannot start your N-task at the specified time, either because system resources are unavailable or because your user task limit is used up, and you have supplied the optional (*RETRY*) parameter, the task will be started as soon as the above restrictions no longer apply.

CHAPTER 5: PREPARING A BATCH TASK

INTRODUCTION

The safest way to check out a batch task is to run it first as a T-task. If it's an autostart task (that is, any B-task, or an N-task other than a split workspace), you could put into ⎕*LX* an expression which invokes a function such as *RESUME* (described in the section entitled "A Sample Latent Expression for a Batch Workspace."), then answer *YES* when the function asks whether you wish to proceed as a T-task.

If it's a split workspace, the easiest way to test it as a T-task is as follows:

1. Set a stop at the statement in your program at which you test, whether the task is the original task or the new N-task. Let the function proceed until that point.
2. By entering a branch statement from immediate execution, transfer control to the statement which, in production, will be the next statement in the spawned task.

In either case, the workspace should be able to do all of its remaining work without asking for further input from the keyboard, and without error. If, while running a batch task, the system encounters a statement that requires input from the keyboard, work on the task is halted, and the workspace is saved in the save-workspace.

While the workspace is running as a T-task, you may seize the opportunity to fix any bugs you encounter. At the same time, note the CPU time and the elapsed time the job requires, to help estimate reasonable limits for the values you must set when you submit the workspace for batch execution. It's also interesting to note the cost of T-task execution, as a baseline for estimating how much you save by running in batch.

OUTPUT GENERATED BY A BATCH TASK

Since a batch task is not connected to a terminal, it can't produce output at the terminal. Output is handled in either of two ways:

- Output which, in a T-task, was explicitly directed to a file, can still be directed there in exactly the same way. This requires no change at all when you convert from a T-task to an N-task or a B-task.

- Output which, in a T-task, was directed to the terminal can be sent instead to a file by using the system function □*OUT*. The procedure is as follows:

  1. Tie a file to which the output can be appended.

  2. Execute □*OUT* **n**, where **n** is a two-element vector whose first element is 0 and whose second is the tie-number of the output file. The first element indicates whether output should also be sent to the terminal. If the first element is 1, output is also sent to the terminal. You might want to use 1 while testing the workspace as a T-task, but during execution of a batch task, output directed to the terminal is simply discarded.

During execution of a batch task, output directed solely to the terminal is lost. Thus, if you fail to use □*OUT*, output statements in the program simply have no effect.

As well, using □*OUT* causes error messages to be transmitted to the output file. The batch task then halts. (This behaviour may be modified through the □*TRAP* event-trapping facility.)

PRINTING DATA PREVIOUSLY SENT TO A □*OUT* FILE

A file generated by □*OUT* is directly acceptable to the system utility *HSPRINT*. You could include a request to have the output file printed as part of the same job that appended the output to the □*OUT* file. Alternatively, the request can be submitted at some later time.

You might subsequently want to print at a terminal the contents of a file written with □*OUT*. If the displays captured in the □*OUT* file were generated simply, there will be one component for each array that the task displayed. But if the program used □*FMT* or □*ARBOUT* to generate its output, the □*OUT* file will contain control messages. You can't just display the control messages; you have to interpret them. To do that, you can use the function *TPRINT* in workspace 1 *FILEAID* to interpret the control codes and generate a readable display. Alternatively, you could use the function *HSP* with the argument '*FILE*' to create a new file (called an H-file) in which only the first component is a control message, and each subsequent component is a character vector (with embedded carriage returns) representing a page of the output.

CHAPTER 6: MONITORING THE PROGRESS OF A BATCH TASK REQUEST

INTRODUCTION

You can check on what the Batch Task Scheduler has done with your batch task request by using one of several functions prepared for that purpose, and stored in the public workspace 1 *BTASKREQ*.

You can check on the progress of a task while it is actually running by using the system function □*RUNS*, whether the task is a T-task, an N-task, or a B-task. Note that any task can, in this manner, check on its own progress. In particular, it is possible to construct a batch task so that it contains internal checks on whether it is progressing as you anticipated.

CHECKING THE STATUS OF YOUR BATCH TASK REQUEST

In addition to the functions *BTASKREQ* and *BTSUBMIT*, described in the Chapter "Submitting a Batch Task Request," the workspace 1 *BTASKREQ* contains several functions for reporting on the status of your requests.

The function *BTREQUESTS* returns as its explicit result a numeric vector containing the request numbers of all requests you have submitted recently. "Recently" means within the last two days, or perhaps longer; the result of *BTREQUESTS* certainly includes any requests that haven't yet been run, no matter when you submitted them. You can **withdraw** a request before it is ever started. If you do that, the request is immediately listed as *WITHDRAWN*.

The status of a request is described by one of the following terms:

QUEUED
The request is in the queue of requests, but has not yet been started.

ACTIVE
The request has been submitted and is running at the moment.

COMPLETED
The batch task was successfully started and has since ended. Note that you can't tell from this whether the task ended because the work was successfully completed, or because an error or interruption caused the work to terminate prematurely.

FAILED
The batch task could not be started. The result of the *BTINQ* function contains information which indicates a specific reason for the failure.

WITHDRAWN
The request was withdrawn via the *BTWITHDRAW* function.

The functions *BTQUEUED*, *BTACTIVE*, *BTCOMPLETED*, *BTFAILED*, and *BTWITHDRAWN* take no arguments. Each returns a vector of request numbers, containing those members of *BTREQUESTS* which have the status indicated by their name. The function *BTLAPSED* returns the numbers of your requests which are in a *COMPLETED*, *FAILED*, or *WITHDRAWN* state.

The function *BTINQ* takes as its argument a vector (or a scalar) of request numbers. Thus, to inquire about all requests, enter:

```
BTINQ BTREQUESTS
```

To inquire only about active tasks, enter:

```
BTINQ BTACTIVE
```

Fields of *BTINQ*

The result of *BTINQ* is a five-column matrix of integers, with one row for each request you inquired about.

Column 1 — the request number field, in which the right argument is repeated.

Column 2 — the request status. The status is represented by an integer according to the following code (in which the value 4 has not yet been assigned a meaning):

0 No such batch task request is on file.

1 The request is queued, but not yet started.

2 The request is active. That is, the requested task has been started but has not yet ended.

3 The batch task has completed. That is, it was successfully started and is no longer in the *ACTIVE* state.

4 The request has been withdrawn.

5 Scheduling status error. This should never arise. If you get a value 5 in this position, please notify your SHARP APL support personnel immediately.

6 The attempt to start the batch task failed. Columns 3 and 4 contain an error code.

Columns 3 and 4 — the run status.

These two columns are used only if the task has started which means that it is either active, completed, or failed. Otherwise, they contain zeros.

When the task is active (status 2), column 3 contains the task-id under which the request is running. Column 4 contains a zero.

When the task is completed (status 3), or failed (status 4), columns 3 and 4 contain a two-element code indicating the outcome of the attempt to run the task. A normal end is indicated by 0 0.

Any outcome other than a normal ending is indicated by a value other than 0 in column 3, and in some instances is amplified by the value in column 4. Codes with value 0 through 17 have the same meaning as those used in the result of *□RUN*, and were listed in the section "The Result of *□RUN*." In addition, other codes specific to batch tasks are listed in the following section, "Results of *BTINQ*."

Column 5 — the task id.
The number of the most recent task started for the request, whether it has completed or not. If the request in question is currently active, this column will contain the same value as in column 3. If no task has yet been started, this value will be zero.

Results of *BTINQ*

These values appear as columns 3 and 4 of the result of an inquiry about the disposition of a batch task request made with the function *BTINQ* in workspace 1 *BTASKREQ*.

56 0 Work was interrupted by a system shutdown, and cannot be restarted because *NORMRST* was not set.

57 0 Work was interrupted by a system crash, but a data error occurred and a restart was not possible.

58 0 The task could not be started because *COPYSP* was specified and the value of the requestor's *□SP* was too large.

59 0 A system crash occurred while the batch task was active and the task could not be restarted because no continue workspace was recovered.

60 0 The task could not be scheduled because a batch task that must precede it (specified by use of the run spec *NBBTASK*) could not be started.

61 0 The task could not be scheduled because a sort that must precede it (specified by use of the run spec *NBSORT*) could not be started or was not successfully completed.

62 0 Work was interrupted by a system crash, and cannot be restarted because *ABNORMRST* was not set.

63 0 The task could not be scheduled because the processing specs were invalid.

DISPLAYING A SUMMARY REPORT OF INQUIRIES

The function *BTDISPLAY* takes as its argument a vector (or scalar) of request numbers, in exactly the same way as *BTINQ*. You might simply supply those numbers directly, for example, in an expression such as

```
      BTDISPLAY 78449 78458
```

Or you might generate the request numbers for the class in which you're interested by using a function such as *BTREQUESTS*, *BTACTIVE*, *BTQUEUED*, *BTFAILED*, *BTCOMPLETED*, or *BTWITHDRAWN*.

The function *BTDISPLAY* does not return an explicit result, but causes printing of a character display containing information similar to the result that you'd get from *BTINQ*, but spelling it out, in full form, rather than stating it as numeric codes. It also shows the processing specs and the □*RUN* parameters that were submitted with each request. For example, suppose you submit a request via *BTSUBMIT* to run a workspace you have called *JOB*1, and at the end to save the active workspace under the name *END*1. The request itself might look like this:

```
      BTSUBMIT ': JOB1 END1 100 6000'
15981
```

The result 15981 is the request number assigned to the request just made. If you now execute *BTREQUESTS*, you get that number, along with the numbers of any other requests that you have submitted recently. The number 15981 would also appear as the result of *BTQUEUED*, but not in the result of *BTCOMPLETED* (which may be empty), since the Batch Task Scheduler hasn't started the task yet.

If you now use *BTDISPLAY* to generate a report on your request 15981, you might see something like this:

```
      BTDISPLAY 15981
BTASK REQUEST     - 15981
REQUESTOR         - 1234567
DATE SUBMITTED    - WED 28 JAN 1986
STATUS            - QUEUED
PROCESSING SPECS  - (NONE)
□RUN PARAMETERS   - : JOB1 END1 100 6000
```

Suppose that, upon seeing this display, you realize that you should have requested that this task start only after work has ended on another request you submitted earlier, for example request number 15040. There's no way to modify the specs once you've submitted them, but it is easy to withdraw a request and submit a new one. Below is an instruction to withdraw two requests:

```
      BTWITHDRAW 15980 15981
0 1
```

The result 0 1 shows that you've successfully withdrawn your request 15981, but that you have had no effect on 15980 (perhaps because it wasn't your request, or perhaps because it is too late to withdraw it since it has already been started). You can now resubmit your request, including an appropriate processing specification, as follows:

```
      BTSUBMIT 'NBBTASK(15040) ⋄ :JOB1 END1 100 6000'
15987
```

Now if you ask to *BTDISPLAY BTREQUESTS*, you'll see both requests, with the first one shown as *WITHDRAWN*:

```
      BTDISPLAY BTREQUESTS
BTASK REQUEST      - 15981
REQUESTOR          - 1234567
DATE SUBMITTED     - WED 28 JAN 1986
STATUS             - WITHDRAWN
DATE WITHDRAWN     - WED 28 JAN 1986
PROCESSING SPECS   - (NONE)
□RUN PARAMETERS    - :JOB1 END1 100 6000

BTASK REQUEST      - 15987
REQUESTOR          - 1234567
DATE SUBMITTED     - WED 28 JAN 1986
STATUS             - QUEUED
PROCESSING SPECS   - NBBT(15040)
□RUN PARAMETERS    - :JOB1 END1 100 6000
```

The possible displays opposite the word *STATUS* are: *QUEUED*, *ACTIVE*, *COMPLETED*, *WITHDRAWN*, *FAILED*, or *ERROR*. If the status of a request is:

ACTIVE - the text (*TASKID NNNN*) will follow the status code *ACTIVE*

FAILED - an error message will follow the status code *FAILED*, for example, *FAILED (LOADWSID NOT FOUND)*, and a line following, indicating the date of request failure

WITHDRAWN - a line will follow, indicating the date the request was withdrawn

COMPLETED - a line will follow, indicating the date the request was completed

WITHDRAWING A BATCH TASK REQUEST

The function *BTWITHDRAW* takes as its argument a vector of request numbers (or a scalar request number) and prevents them from being run.

The result of *BTWITHDRAW* is a boolean vector having the same length as the argument. It contains a 1 for each task that was successfully withdrawn, or 0 otherwise.

You can only withdraw a request that you submitted. Where a number in the argument of *BTWITHDRAW* refers to a task that was submitted by someone else, or has already started, or does not exist, the corresponding element in the result of *BTWITHDRAW* is 0.

After a request has been withdrawn, an inquiry about that request (using *BTINQ*) shows the value 4 (withdrawn) in column 2, and 0 0 (successful) in columns 3 and 4.

CHECKING THE STATUS OF AN ACTIVE TASK

The system function □*RUNS* describes the current situation of an active task. It tells you how long the task has been running and how many CPU units it has used.

A B-task, while it is running, is indistinguishable from an N-task. Thus, for either kind of task, the only way to monitor its progress while it is actually running is to use □*RUNS*. That can be done either by having the task monitor its own progress, or by having some other task (for example, your terminal task) inquire about the status of the task in question.

The system function □*RUNS* provides a report on the status of all of your tasks active at the moment. □*RUNS* gives you a matrix describing all of the tasks that meet **either** (or both) of two criteria:

- The task are running on your account (regardless of who started them)
- The tasks are running because you requested them, regardless of the account under which they are running.

For one task to check on the activity of another, or of itself, by using □*RUNS*, three things are necessary:

- The task you're checking must be one that you own, either because you started it yourself or because someone else started it on your behalf.
- The task that you're checking must actually be running as you check.

- If several tasks are running at once, you'll need to know their task-ids, in order to know which row of the matrix produced by □*RUNS* refers to which task.

□*RUNS* is a niladic function. That is, it takes no argument. It gives you a matrix result, with one row for each task. The first row always describes the current task. If you're simply signed on at a terminal without using any non-terminal tasks, that matrix has just one row.

Columns in □*RUNS*

Column 1: The task number (task-id).

Column 2: The account number of the task's owner.

Column 3: The account number of the owner of the task that started it. Note that your account number must appear at least once on every row, either in column 2 or in column 3, or in both.

Column 4: The type of task: 0 for T-task, 1 for N-task, and 2 for B-task.

Column 5: The amount of CPU time so far consumed by the task (in CPU units).

Column 6: The number of connect seconds so far consumed by the task.

Column 7: The maximum number of CPU units that the task is permitted.

Note: This appears as 0 for a T-task, as terminal tasks have no CPU or connect limit. However, you may set an upper limit for N-tasks and for B-tasks as part of the procedure that starts them.

Column 8: The maximum number of elapsed seconds that the task is permitted (see the note for column 7).

Sample Use of □*RUNS*

Below appears an example from a session in which no batch tasks were running, and so □*RUNS* returned only one row:

```
      □RUNS
4906  1234567  1234567  0  114  6402  0  0
```

The first element shows that the task being described is task 4906. The fact that the fourth element is 0 means that this is a T-task, and because it is a T-task, elements 7 and 8 are also zero.

Element 2 shows that the task is owned by user 1234567, and element 3 shows that the task was started by user 1234567. Since

this is a T-task, elements 2 and 3 must contain the same value, because no user can start a T-task owned by someone else.

Element 5 shows that the task has consumed 114 CPU units, and element 6 shows that 6402 seconds (about an hour and three quarters) have elapsed since the task was started.

If, a short time later, user 1234567 starts an N-task, □*RUNS* shows both the terminal task (with the same task-id as before) and the N-task:

```
      □RUNS
4906  1234567  1234567    0    119   8284   0    0
5137  1234567  1234567    1      4    520  10  600
```

The first row of the result always refers to the task that made the inquiry. In the example above, the remaining row shows the N-task. However, after the first row, the order is unpredictable. You can't use the order in which the other tasks appear in the rows of □*RUNS* to clarify which task is which. The task-id of the N-task is 5137. Like the T-task, it is owned by 1234567 and was started by 1234567. Its task type is 1, meaning that it is an N-task. It was allotted a maximum of 10 CPU units, and has used 4, and a maximum of 600 connect seconds, of which 520 have elapsed.

If user 1234567 now submits an N-task on behalf of account 3344555, and that N-task receives the task-id 5148, the result of □*RUNS* might appear as follows:

```
      □RUNS
4906  1234567  1234567    0    119   8344   0    0
5137  1234567  1234567    1      4    580  10  600
5148  3344555  1234567    1      1     20  10  600
```

For task 5148, the values in columns 2 and 3 are now different, since the task is owned by user 3344555, but was started by user 1234567.

If at the same time a task owned by user 3344555 was to execute □*RUNS*, that task would see itself, and also the same row describing task 5148 (because 3344555 is its owner).

IDENTIFYING THE TASK-ID OF THE ACTIVE TASK

The task-id of the active task is always the first column in the first row of the matrix produced by □*RUNS*.

Alternatively, a task can discover its own task-id directly from one of the uses of the system function, □*WS*. The expression

```
2 □WS 3
```

yields a numeric vector describing the workspace environment, in which the task-id is the eleventh element; that is:

```
(2 ⎕WS 3)[⎕IO+10]
```

INTERRUPTING A B-TASK OR AN N-TASK

The system function ⎕*BOUNCE* terminates an active task. It takes as its argument a vector (or a scalar) of task-ids. To interrupt a B-task or scheduled N-task, the appropriate argument to ⎕*BOUNCE* is not its request number, but the task number it received when the Batch Task Scheduler started it.

If the tasks appearing in the argument of ⎕*BOUNCE* are active and are owned or initiated by the account that executes the ⎕*BOUNCE*, each task is terminated as if by a **strong interrupt**. That is, while the interruption probably takes place at the end of the execution of a line, if a particular function within that line consumes a lot of CPU time, the line may be interrupted before its completion.

If the task thus terminated was an N-task or a B-task, the active workspace is saved in the save-workspace (just as it would be if work were interrupted by a system shutdown or by the expiration of the connect-time or CPU limits).

⎕*BOUNCE* can also be used to terminate the user's T-task. In that case, the active workspace is saved under the name *CONTINUE*, as it would be if work were terminated by a communications failure or a system shutdown.

The result of ⎕*BOUNCE* is a boolean vector of the same shape as the argument, containing a 1 for each task actually terminated, and zero otherwise.

ACCOUNT INFORMATION ⎕*AI*

The system function ⎕*AI* returns account information. The first element is the account number of the owner of the task. The second and third elements are CPU time (in milli-CPU-units) and connect time (in milliseconds) for the current task. For N-tasks and B-tasks, these first three elements are the only values reported; the other elements of ⎕*AI* are zero.

For T-tasks, ⎕*AI* also includes keying time, terminal I/O characters, and cumulative values of connect time, CPU units, keying time, and characters since the date of the last billing. Those cumulative values are available only from a T-task, and refer only to work accumulated during T-tasks. They included no allowance for usage by N-tasks and B-tasks you may have run.

If you need to analyse time spent in N-tasks and batch tasks, consult the workspace 1 *USAGE*, described in the "Usage Inquiry System User's Guide".

FACTORS IN THE DESIGN OF THE LATENT EXPRESSION

The latent expression is what starts the work in all B-tasks, and in all N-tasks except those that start from a split workspace. So $\square LX$ must be set with care. But that isn't the only factor. Consider the following list of points at which $\square LX$ has an effect:

Start of the batch task
: For all B-tasks, and for all N-tasks other than those that started from a split workspace, $\square LX$ is the only means of providing the initial instruction to start all the work that the task will perform.

End of the batch task
: At the end of every task, the active workspace is saved. That includes both split N-tasks and autostarted N-tasks, and it includes both normal endings and endings forced by an error or interruption.

  When the workspace is saved, it contains a latent expression. When you subsequently load the save-workspace, the latent expression with which it was saved will (as usual) be executed automatically. Unless you have changed $\square LX$, that will be the latent expression that started the job. Or if the task was run as a split workspace, it may be whatever $\square LX$ was in the workspace at the time it was split. When the save-workspace is subsequently reloaded, you don't want the work to start again.

Resumption of a batch task that's been interrupted
: If an N-task or a B-task is interrupted before its completion, its active workspace is saved. Later on, that saved workspace may be loaded, perhaps because you want to figure out what went wrong, because the job was a batch task request for which you have permitted automatic restarting, or because you are ready to restart the work after diagnosing the reason it was interrupted. When the saved workspace is reloaded, either to resume work or so you can diagnose what went wrong, you don't want it to immediately start work over from the beginning. You need a latent expression that won't get in the way of your trouble shooting (when the workspace is reloaded during a T-task), but that still permits the system to resume work in a reasonable way (as a B-task or an N-task).

Files that were in use when a batch task was interrupted
: When a batch task is interrupted, whatever files it had tied are automatically untied. Although the save-workspace that is then saved does contain the execution stack and line counter (so the program can be resumed where it left off), it does not contain any automatic record of the names of files that were tied

(□*NAMES*) or of the numbers used to tie them (□*NUMS*). Thus, although the program can resume exactly where it left off, it will be doing so without its files unless explicit steps are taken to retie them.

Testing and development
During development of the workspace, or when modifications are made to it after it has been completed, you'll probably want to load it to make the changes. At that time you probably won't want it to start working immediately. So for this reason also it is desirable to have a latent expression that starts work only at the beginning of the real task: that is, when it is either an N-task or a B-task, but not when the workspace is reloaded as a T-task.

INGREDIENTS OF THE LATENT EXPRESSION FOR A BATCH TASK

Based on the factors mentioned in the preceding section, here's a list of features desirable in the latent expression for a batch task.

- The same latent expression should cover all the situations in which it may be executing:
  - Normal start of a batch task.
  - Testing of the batch task as a T-task.
  - Diagnosis of save-workspace if an error interrupts work.
  - Resumption of batch execution after interruption.
  - Reloading of the save-workspace after a normal end.
- The latent expression should be a branch; for example, something in the form →*RESUME*. In order to resume work after interruption, the latent expression must be a branch statement (i.e., one that starts with the branch arrow →) because no other type of APL statement can cause resumption of a suspended function. The branch instruction can be constructed so that, when appropriate, it also takes care of the steps required in the various other situations.
- The destination (*RESUME* in the preceding example) should be a function which, as a side effect, takes the needed steps (starting the main function, or retying files, as appropriate) before it returns the branch destination as its result.
- The function invoked in the latent expression should discriminate between the different circumstances under which it may have to work.
  - Is the current task a batch task or a T-task? If it's a T-task, the workspace has probably been loaded for testing or inspection, and so work shouldn't start automatically.
  - Is the state indicator empty (except for the latent expression itself), or is execution suspended somewhere within the main task?

    If the state indicator contains functions other than the latent expression itself, work on the main function has probably been interrupted, and should be resumed, after files are retied.

But if the state indicator is otherwise empty, either work hasn't yet begun, or else work has been successfully completed. The main function should provide a way to decide which.

A SAMPLE LATENT EXPRESSION FOR A BATCH WORKSPACE

The following example isn't intended as definitive; it merely illustrates some issues and some possibilities.

Suppose you set up a workspace to be run in batch in which, as a matter of convention, the expression to be executed in order to get your work done is always stored as a character vector having the name *EXPRESSION*. For example, if the job you want done can be evoked by the expression *LEAVEACCRUAL 1 178 TO 3 31 78*, then you set that up by entering:

```
EXPRESSION←'LEAVEACCRUAL 1 1 78 TO 3 31 78'
```

Suppose further that the latent expression in your batch workspace is *→RESUME EXPRESSION*. You make that the latent expression by entering:

```
      ⎕LX←⍞
→RESUME EXPRESSION
```

Starting the main program, or resuming work appropriately if it has been interrupted, is handled by the function *RESUME*, working with the expression *⍎⎕LX* (which the system always executes automatically whenever a workspace is loaded).

The sample function *RESUME* incorporates ways of handling each of the issues mentioned in the preceding section. Its definition follows.

```
    ∇ LC←RESUME EXPRESSION;TASKTYPE
[1]   TASKTYPE←⎕RUNS[⎕IO;⎕IO+3]
[2]   →(TTASK,NTASK,BTASK)[0 1 2 ⍳TASKTYPE]
[3]   ⍝ ⍟ SHOULD NEVER REACH HERE
[4]  TTASK:'SAY WHETHER YOU WISH TO EXECUTE' ◇ EXPRESSION
[5]   →(((2 3 ⍴'YESNO ')∧.=3↑⍞)/START,ABORT),TTASK
[6]  ABORT:→ ⍟ ABORT ⎕LX BRANCH, LEAVING STACK AS BEFORE
[7]  BTASK:
[8]  NTASK:→(2≥⍴⎕LC)↑START
[9]   FNAMES RETIE FNUMS
[10]  LC←⎕LC SELECT 2 ⎕WS 2
[11]  →0
[12] START:⍎EXPRESSION
[13]  ⎕LX←'''',EXPRESSION,' COMPLETED ',(⍕⎕TS),''''
[14]  → ⍟ AVOID BRANCH IN ⎕LX
    ∇
```

Lines 1, 2 and 3
: The function tests whether its environment is a T-task, an N-task, or a B-task.

Lines 4, 5 and 6
If it is a T-task, the function displays the expression and asks whether you want to go ahead with executing it. If you reply *YES* (presumably because you want work to start from the beginning), the function goes ahead with executing it. Otherwise, the function does nothing further, but aborts, leaving the state indicator as it was before *RESUME* was executed.

Line 7
In this definition of *RESUME*, an N-task and a B-task are treated in the same way. Clearly, you could treat them differently if you preferred. Because here line 7 is empty, control falls through to line 8.

Line 8
If there is nothing on the stack but *RESUME* itself and the ⍎⎕*LX* which called it, this must be an attempt to start execution of the batch workspace. Conversely, if there is something else on the stack, this must be an attempt to resume execution in a workspace saved before execution had been completed. Line 8 checks only whether the length of ⎕*LC* is less than or equal to 2; if desired, it could more specifically check the names of the functions on the stack by examining 2 ⎕*WS* 2. The function ⍎ increases the length of ⎕*LC* by one during its execution. Thus, during start or restart, ⎕*LC* will have at least two elements, the current line number of *RESUME* and a zero for the ⍎ in ⍎⎕*LX*.

Lines 9, 10 and 11
These lines are executed only while trying to restart the task after it has been interrupted.

The defined function *RETIE* (definition not shown) reties the needed files. It is assumed that the application program maintains a current list of the files it has tied and the tie numbers used to tie them, in the global variables *FNAMES* and *FNUMS*.

The defined function *SELECT* (definition not shown) calculates the line number of the line to be executed next. Before returning its result, it may consult the state indicator (2 ⎕*WS* 2) and the line counter (⎕*LC*), to determine whether, considering the point at which execution was halted, some additional measures need be taken. To write such checks into *SELECT* would require detailed knowledge of the application evoked by ⍎*EXPRESSION*. If any such additional steps are taken, they are side-effects of *SELECT* and are carried out before *SELECT* returns as its result the number of the line next to be executed. Note that whatever result is returned, it can only produce a branch to a line **within the most recently-called function**, or else an exit from the most-recently called function, since APL does not permit any other sort of branching.

Line 12
This calls for execution of *EXPRESSION*, thereby evoking the desired application program.

Line 13
When work has reached a normal end, the latent expression is reset so that it contains only a message showing what was done and when it was completed. Then execution is terminated.

Line 14
Execution is terminated by use of → to avoid the branch contained in the original *⎕LX* value.

DETERMINING WHY A BATCH TASK TERMINATED

A convenient method for determining why a batch task terminated is to examine the value of the system variable *⎕ER* in the saved workspace. Often this can be done without loading the workspace by copying the value of *⎕ER* from the saved workspace into your active workspace. Of course, this will not work if the saved workspace contains suspended functions which localize *⎕ER*, since *)COPY* copies global objects only. In this case, the saved workspace must be loaded if the local *⎕ER* value is to be examined. Care should be taken in the design of the recovery latent expression so that the value of *⎕ER* is not altered by the recovery mechanism, particularly when a T-task loads the saved workspace.

FILE SYNCHRONIZATION FOLLOWING A SYSTEM CRASH

Like other computer systems, SHARP APL is occasionally subject to hardware or software failures which bring the entire system to a halt for an interval ranging upwards from a few minutes. Such an interruption of service is referred to as a "system crash." Past performance suggests that if the elapsed time a task requires reaches as much as one hour, there is about a one percent chance that the task will be interrupted by a crash.

The procedures used internally by the file system ensure that any attempt to write a file component either goes to a successful completion, or, if interrupted, leaves the file as it was before the writing was attempted. So there's very little risk that a file itself is defective following a system crash. The risk is that, although the file is intact, the recovered workspace is not quite in synchronization with it.

Following a system crash, it is almost always (although not universally) possible to recover each task's active workspace. The active workspace of a T-task is saved under the name *CONTINUE*. The active workspace of an N-task or a B-task is saved in the save-workspace. However, it is possible that a workspace thus recovered dates from a moment to a few seconds earlier than the time of the crash itself. On the other hand, any files that the task was using will contain every change actually completed right up to the instant of the crash. This means that it is possible that the files contain information slightly more recent than the recovered workspace.

That discrepancy in timing can arise for the following reason. After the crash, it may prove impossible to recover the active workspace

of a few of the tasks which were active. For those tasks, the system then falls back on a set of emergency copies of all active workspaces that (as a precaution) it makes at least once every 4 seconds (and under some conditions more frequently). But nevertheless this means that if your task is one of those for which the very latest active workspace cannot be recovered, you may end up with a recovered workspace that represents the way things were as much as 4 seconds earlier than the time of the crash.

A task usually maintains in the active workspace some record of its progress, such as variables indicating the particular file components it is currently writing. But if the workspace is out of date by a few seconds, those values could be pointing to file components that in reality have already been processed.

How serious a problem this presents depends upon how the application is written. In many cases, it means only that a few reads and writes to the file may have to be repeated when the task is resumed. But restarting could lead to problems if the work involves updating file components so that the new value is a function of the old value of the same component. For example, suppose the program were to go through a file, and, for each component, read it, add a constant, and then write it back. Following a restart with an out-of-date counter, some of the components might have the constant added twice.

To avoid troubles of that sort, it's important to design the application so that, if need be, the program can deduce from examination of the file itself how far along the task was when work was interrupted. There are several ways to do that. One approach is to keep such pointers in the file as well as in the workspace, and then make sure that upon restart the file values are read back into the workspace. Another is to write into each component itself some indication that processing has been completed. Or you could deduce the same thing by comparing the time at which the active workspace was last saved with the result of `⎕RDCI`, the timestamp which is written for each component of a file. It is prudent to avoid altogether any technique in which you replace a component by a value which depends solely on the value previously written in the same place.

Note on alphabetization: The right parenthesis,), is alphabetized as though it didn't exist; thus, *)COPY* is under **C**. "⍤" is alphabetized as though it were spelled "jot", "⍎" as though it were spelled "hydrant", and "⍞" as though it were spelled "quad".

A

ABNORMST 17-18 26
Abnormal restart 17-18 26
ACTIVE status of request 23
Account information, obtaining with ⎕*AI* 31
Account number
  association with task 7-8
  in result of ⎕*RUN* 13 29-30
  specifying in control statement 9
Active task
  checking the status of 28
  identifying task-id 30
  terminating 31
Active workspace
  ownership of 7
  split, testing the result of ⎕*RUN* 12
Advantages and disadvantages. **See** Batch task
Arguments for ⎕*RUN*, sample 11
ATTN (weak interrupt) 17
Autostart task
  and latent expression 33
  running as a T-task 21
Autostart, starting a batch task with 3-4

B

B-task
  description of 3
  files tied by 5
  maximum number permitted 7
  ownership of 7
  passing initial data to 4
  starting with ⎕*RUN* 9
Batch task
  advantages and disadvantages 1-2
  completion of 3 10 33
  error messages 22
  errors 1-2 10-11
  files 2 21-22 33-34 37-38
  general overview 1
  input 2-4 21
  inquiring and monitoring 12-13 23-32
  interruption 10-11 17-18 31 33-34
  non-terminal. **See** N-task

operator intervention 19
output generated by 21-22
ownership of 7 29
processing specifications 15-16
ABNORMRST 17
COPYSP 19
NBBTASK 18
NBDATE 18
NBOR 19
NBSORT 19
NBTIME 18
NBTS 18
NORMRST 17
NTASK 19
request number 15
request, information needed to submit 15-20
restarting 33-36
scheduler
behavior with high connect limit 10-11
inquiring on status of a request 15
starting a B-task 3
submitting B- or scheduled N-task requests 9
scheduling 2-3 18-19
starting 3-4 7-8 9 15-16 33-37. **See** also Starting a batch task
status 23-24
termination 2-5 10-11 17
testing 21 34
withdrawing 24 28-29
Billing. **See** Account information
Bounce. **See** □*BOUNCE*
BREAK (weak interrupt) 17
BTASKREQ function
description of 15
sample request using 16
BTASKREQ workspace 9 13 23
BTCOMPLETED function 24
BTDISPLAY function, generating a report on a request 26-28
BTFAILED function 24 26-27
BTINQ function 24-25
BTLAPSED function 24
BTQUEUED function 23-24
BTREQUESTS function 23 26-27
BTSUBMIT function 15-16 26-27
BTWITHDRAW function, description of 27-28
BTWITHDRAWN status 23-24 26-27

C

CLEAROUT function 10
CONTINUE workspace 10
COPYSP 19 25
Colon, in argument to □*RUN* 9 11
Communication between parent and N-task 5
Connect limit
estimating from T-task usage 21
in argument to □*RUN* 9-11
in relation to *NORMST* 11
in result of □*RUNS* 30-31
Connect seconds, determining amount consumed 29

Control statement 9
)*COPY*, in consideration of □*ER* 37
CPU limit, in argument to □*RUN* 10
CPU time, determining amount consumed 29
CPU units, determining maximum permitted 29
Crash, system. **See** System crash
!h08
D
Date, in processing specs 18 26-27
Disadvantages of batch tasks 1

E
Elapsed seconds, determining the maximum number permitted 29
Elided sign-on, indication of 11
Endless loop 11
Error in argument to □*RUN* 10
Error messages 22
Execution stack, in save-workspace 33-34

F
FAILED status 24
Failure, system. **See** System crash
FILEAID workspace 22
File synchronization 37-39
Files
    retying at restart 36-37
    tied by N- and B-tasks 5
Functions in public workspace 1 *BTASKREQ*
    automatic 15
    interactive 15

H
Highspeed printing 1
HSPRINT system utility 22
⍟ 36-37
H-file 22

I
Immediate execution mode, effect on N-task 4-5
Infinite loop 11
Input to batch task. **See** Batch task
Inquiring on a batch task. **See** Batch task
Interruption of a batch task. **See** Batch task
Invalid sign-on 13

J
∘, in argument to *BTSUBMIT* 16

L
Latent expression. **See** □*LX*
Leading zeroes, in time of day processing specifications 18
Limit
    CPU. **See** CPU limit
    connect. **See** Connect limit
Loading a workspace to start a task 9
Locked out 13
Locked workspace 9
Loop, endless 11

N
NBBTASK option 18-19
NBDATE option 18
NBOR option 19
NBSORT option 19
NBTIME option 18
NBTS option 18
NORMST option 10-11 16-17
Non-terminal task. **See** N-task
Normal restart 17
Number, request 15 23-24 26
N-task
  communication with other tasks 5
  direct, description of 2
  distinguishing from parent task 7
  files tied by 5
  maximum number permitted 7
  ownership of 7
  passing initial data to 4
  scheduled, description of 3
  starting with □*RUN* 9

O
Output, generated by a batch task 2 21-22
Ownership of a batch task 7
Ownership of non-terminal tasks 7

P
Parameters, □*RUN* 16
Parent task, communication with N-task 5
Passing initial data to an N-task or B-task 4
Password, in argument to □*RUN* or *BTASKREQ* 7 13
Printing output 22
Processing specifications. **See** Batch task 15

Q
□*AI* function, returning account information using 31
□*ARBOUT*, effects on generating output 22
□*BOUNCE* 10 31
□*ER*, determining reason for task termination 37
□*FMT*, effects on generating output 22
□*LC* checking length of 36
□*LX*
  branch 34
  features desireable in 34-35
  issues in design of 33-34
  sample for a batch task 35
  setting it up to autostart a batch task 3-4
  use in testing a batch task 21
□*OUT*, sending output to a file or to print 22
□*RDCI*, use of to synchronize file after a "system crash" 37-38
□*RUN*
  checking the status of an active task 28
  columns in result of 29
  fields 9-11
  indication of an invalid argument or failure 12
  indication of failure or invalid argument 12
  parameters 16

result of 12
sample arguments for 11
sign-on 9 11
testing the result of, after splitting the active workspace 12
status 24-25
use to start N-task 9
values that can result from 12-13
□*RUNS*
checking progress of an active task 23
columns in result of 29
sample uses 29-30
status 28
□*SP*
passing initial data to an N- or B-task 4
propagating the value of, into a B-task workspace 19
□*TS* system function, timestamp format resembles 18
□*WS* function, identifying task-id 30

R
Request number 15 23-24 26
Request status
ACTIVE 23
COMPLETED 23
FAILED 24
QUEUED 23
WITHDRAWN 24
Restarting. **See** Batch task
Running a B-task as an N-task 19

S
Save-workspace name, specifying in control statement 10
Scheduling. **See** Batch task
Shared variables, use of 2-3
Shutdown, normal versus abnormal 17
Sign on
in argument to □*RUN*. **See** □*RUN*
indication of elided 11
Sort, required before batch task starts. **See** Batch task, operator intervention
Spawned workspace. **See** Split workspace
Split-workspace 33
starting a batch task with 4
testing as a T-task 21
testing the result of □*RUN* 12
Starting a batch task
with autostart workspace 3
with split-workspace 4
specifying a load-workspace 9
specifying a save-workspace name 10
specifying CPU limit 10
specifying connect limit 10
specifying the account number 9
Status of request. **See** Request status
Status, of an active task, checking 28
Strong interrupt 31
Submitting a B-task request
with *BTASKREQ* 15
with *BTSUBMIT* 15

System crash
file synchronization following 37-38
restarting a batch task. **See** *ABNORMRST*
S-task, description of 2

T
Task
checking progress of an active, using □*RUN* 23
checking status of an active 28
communication between 5
determining why it terminated 37
four main kinds 2-3
indication of successful start 12
interrupting using □*BOUNCE* 30
items associated with each 8
maximum number permitted 7
obtaining account information (using □*AI*) 31
request
generating summary reports of (using *BTDISPLAY*) 26-27
obtaining list of all recent (using *BTREQUESTS*) 23
withdrawing 27-28
restarting after abnormal shutdown 17
restarting after normal shutdown 17
starting. **See** Starting a batch task
type, in result of □*RUNS* 29
use of control statement in submitting non-terminal 9
Task-id, identifying 30
Task-id, in result of □*RUNS* 29
Task-id, indication of task owner 7
Termination. **See** Batch task
Testing. **See** Batch task
Time-stamp
file component 38
in processing specs 18
TPRINT function, interpreting control codes in an output file 22
T-task, description of 2

U
USAGE workspace 31
User account. **See** Account number
User number. **See** Account number

W
Weak interrupt 17
WITHDRAWN status. **See** Batch task
Withdrawing a batch task request 27-28
Work session 2
Workspace name, loading to start a task 9
Workspace 1 *BTASKREQ*
BTACTIVE function 24
BTCOMPLETED function 24
BTFAILED function 24
BTQUEUED function 24
BTWITHDRAWN function 24
use of functions in 24
Workspace 1 *FILEAID*, interpreting control codes in an output file 22
Workspace

active 1-4 28-31 **See** also Active workspace
batch 5
clear 10
environment described by □*WS* 30-31
latent expression 3-4 13
library 7 9-10 17 33-35
limits 10-11 13
load 3-4 9-10
name
    in argument to □*RUN* 11-12
    load 3-4 9-10 13
    save 2-3 10 33
names. **See**
BTASKREQ, FILEAID, HSPRINT, USAGE, CONTINUE

Z
Zeroes, leading, in time of day processing specification 18